Measuring the Structural Quality of Business Applications

Bill Curtis  
CAST  
PO Box 126079  
Fort Worth, Texas 76126  
curtis@acm.org

Jay Sappidi  
CAST  
373 Park Ave. South, 5th Floor  
New York, NY 10016  
j.sappidi@castsoftware.com

Jitendra Subramanyam  
CAST  
373 Park Ave. South, 5th Floor  
New York, NY 10016  
j.subramanyam@castsoftware.com

Abstract—This study summarizes results of a study of the internal, structural quality of 288 business applications comprising 108 million lines of code collected from 75 companies in 8 industry segments. Results are presented for measures of security, performance, and changeability. The effect of size on quality is evaluated, and the ability of modularity to reduce the impact of size is suggested.

Keywords—Software metrics, Software quality, Benchmarking, Static analysis, Internal quality, Maintainability Performance efficiency.

I. INTRODUCTION

The purpose of this study is to provide an objective, empirical foundation for evaluating the internal, structural quality of application software in both the public and the private sector. Such studies are needed to help IT organizations make visible the costs and risks hidden within their application portfolio, as well as be able to identify the factors that most affect quality. Internal quality involves the non-functional, internal properties of an application. It evaluates the engineering soundness of an application’s architecture and coding, rather than the correctness with which the application implements functional requirements.

Internal quality characteristics are critical because they are often difficult to detect through standard testing, yet are frequent causes of operational problems such as outages, performance degradation, breaches by unauthorized users, and data corruption [1]. Internal quality metrics have been shown to correlate with criteria such as maintenance effort and defect detection [2]. The first enumeration of such quality characteristics was provided by Boehm and his colleagues at TRW in the 1970s [3].

II. THE SAMPLE AND DATA

The data in this paper are drawn from 288 business applications, representing 108 M lines of code (3.4 M Backfired Function Points), submitted by 75 organizations for static analysis of their structural quality characteristics. The results of these analyses are captured in AppmarQ, a structural quality benchmarking repository maintained by CAST. These 75 organizations represent 8 industry segments including banking, insurance, manufacturing, telecommunications, energy, IT consulting, software ISVs, and government. These organizations are located primarily in North America, Europe, and India. The languages in which these applications are written include COBOL, Java EE, .NET, C, C++, and ABAP. The applications range from 10,000 to 5 million lines of code (LOC), with a mean of 374,220 LOC. Of these applications, 26% contain less than 50,000 lines of code, 32% fall between 50,000 and 150,000 thousand lines of code.

Since there is no rigorously developed population description of the global trove of business applications, it is impossible to assess the generalizability of these results. Although these results may not characterize the global population of IT business applications, they do emerge from what is believed to be the largest sample of applications ever to be statically analyzed and measured against internal quality characteristics across different technologies. Because of the selection process for submitting applications to deep structural analysis, we believe this sample is biased toward business critical applications.

These business applications were analyzed using CAST’s Application Intelligence Platform (AIP) which performs a static analysis of an entire application using over 900 rules to detect violations of good architectural and coding practice. These rules have been drawn from an exhaustive study of software engineering texts, online discussion groups focused on application best practices and defects, and customer experience drawn from defect logs and application architects.

The AIP begins by parsing an entire application at build time to develop a representation of the elements from which the application is built and its data-flows. This analysis is normally performed at during the build in order to analyze the source code at the application level across various language and technical platforms. The AIP includes parsers for the 28 languages such as J2EE, .NET, C, C++, PHP, ABAP, SQL, COBOL, and a universal analyzer that provides and 80% parse for languages lacking a dedicated parser. Once parsed, AIP looks for violations of its architectural and coding rules and identifies the number of violations versus the number of opportunities for violations for each rule. The results are aggregated to the application level where each violation is weighted by its severity and summed into both a health factor score for a thousand lines of code.

DEFINITIONS

Static analysis, Internal quality, Maintainability Performance efficiency.
well as a Total Quality Index that includes all violations. AIP provides a series of management reports and tools that guide developers in drilling down to specific violations that need remediation.

The application health factors in AIP were selected after reviewing ISO 9126, a standard describing software quality attributes. However, since the quality characteristics in ISO 9126 have not been defined down to a level that can be computed from the source code, some health factors names differ from those in ISO 9126 based on the content analyzed and the meaningfulness of the names to users of the technology. In order to provide more standardization to computable measures of internal quality, the senior author is leading an industry effort called the Consortium for IT Software Quality sponsored by the Software Engineering Institute at Carnegie Mellon University and the Object Management Group that is developing standard definitions for automatable metrics. CISQ intends to make these metrics as consistent as possible with the emerging ISO 25000 series of standards which will replace ISO 9126.

This paper will concentrate on results for three structural quality characteristics—security, changeability, and performance efficiency. The number of rules evaluated for each of these quality characteristics ranged between 176 and 506. Scores for each of these internal quality characteristics are aggregated from the component to the application level and reported on a scale of 1 (high risk) to 4 (low risk), using an algorithm that weights the severity of each violation and its relevance to each individual quality characteristic. The scores reported here represent the frequency with which architectural and coding best practices related to each quality characteristic have been violated in this sample of applications.

III. SECURITY

Security scores evaluate the attributes of an application that affect its ability to prevent unauthorized intrusions and loss of confidential data. The distribution of Security scores across the sample is presented in Figure 1. The bi-modal distribution of Security scores suggests that applications there are group differences in the Security aspects of business applications.

Further analysis revealed that applications with higher Security scores were predominantly large mainframe-based COBOL applications in the financial services sector (banking and insurance) which differed significantly from the scores for other languages (Figure 2). In these industry segments high security for confidential financial information is mandated. Mainframe, COBOL-based applications are also less exposed to the security threats that challenge Web-facing applications.

Figure 1. Distribution of Security scores.

Figure 2. Security Scores by Language.

The lower Security scores for other types of applications are surprising. In particular, .NET applications received some of the lowest Security scores. These data suggest that the IT community’s attention to security may be primarily driven by compliance regulations within industry segments. However, the higher density of security-related violations in .NET is a concern for Web-based applications where personal or confidential information could be at risk.

IV. CHANGEABILITY

Changeability scores evaluate the attributes of an application that make it easier and quicker to modify. Changeability scores presented in Figure 3 exhibit a bi-modal distribution indicating important differences in the maintainability of applications across the sample. Since the Changeability of an application is a critical driver of its cost
of ownership, this distribution suggests large differences in the costs of ownership between applications with high and low Changeability scores.

![Figure 3. Distribution of Changeability scores.](image)

We compared Changeability scores by industry segment. The results revealed that scores for applications in the public sector are significantly lower than those in other segments (p < .01). Our sample included government applications from both the US and EU. Although we do not have application cost data, these results suggest that government agencies are spending significantly more of their IT budgets on maintaining existing applications than on creating new functionality. Not surprisingly, in their 2009 IT Staffing & Spending Report Gartner reported that the government sector spends about 75% of its budget on maintenance, higher than any other segment.

Poor Changeability scores in the government sector may be partially explained by the high percentage of outsourced applications in the public as compared to the private sector. Seventy-five percent of the government applications in this sample were outsourced compared to 51% for the private sector. When government applications were removed from the sample, there was no significant difference between the Changeability scores for insourced and outsourced applications.

The lower Changeability scores for government agencies may partially result from conditions unique to their acquisition. Multiple contractors working on an application over time, disincentives in contracts for building easily maintained code, and challenging acquisition practices are potential explanations for lower Changeability scores for government applications.

V. PERFORMANCE

Performance scores evaluate the attributes that affect the responsiveness and efficiency of an application. Performance is usually evaluated through measuring the execution behavior of the application dynamically. However, static analysis is able to detect violations of good coding practice, such as instantiating objects inside loops, that can cause performance problems as the database grows or the user increases. Scores for Performance were widely distributed as displayed in Figure 4, and in general were skewed towards better performance. However, Performance exhibited the widest variation of the distributions of any of the internal quality characteristics in this study.

![Figure 4. Distribution of Performance scores.](image)

The skew toward higher scores observed for Performance might be explained by hypotheses that involve both technology and people factors. First, the availability and use of automated performance testing solutions has made performance problems easier to detect and address during development. Second, performance problems are experienced immediately by users whose productivity it impacts. It is not uncommon for end-users to complain vociferously to the development team about slow performance, prioritizing the remediation of performance problems over other quality problems such as poor maintainability.

Performance scores are displayed by language in Figure 5. The results indicate that the lowest median scores for Performance were achieved by J2EE and .NET. Since COBOL applications are the oldest in the sample and have been adjusted for performance issues for well over one or
two decades, it is not surprising that their Performance scores are the highest. However, the lower scores for J2EE and .NET lead to several hypotheses regarding such factors as the awareness of younger developers of performance issues in the architecting of applications, the structure of the newer languages, the improved use of modularity in designing applications, and other such factors. These hypotheses will receive further investigation.

![Figure 5. Performance Scores by Language.](image)

**VI. EFFECT OF SIZE ON QUALITY**

The relationship between size and quality in this sample was investigated by combining the various quality characteristics into a combined Total Quality Index and correlating it with application size. The result for the sample as a whole contradicts the conventional wisdom that the quality of an application decreases as it grows larger. The correlation between the Total Quality Index and size was insignificant ($r < .01$), as were the correlations between each of the separate quality characteristics and size.

However, when investigated within different language categories, the Total Quality Index did correlate negatively with the size of COBOL applications ($r = -.67$, $p < .01$). One possible explanation for this negative correlation is that COBOL does not encourage modularity, resulting in applications possessing many large and complex components. The design of more recent languages encourages modularity and other techniques that mitigate the affect of complexity as applications grow larger.

To investigate this explanation we compared the percentage of highly complex components across different languages. For the purpose of this analysis we defined a highly complex component as having high scores on measures such as Cyclomatic Complexity and coupling. The percentage of highly complex components in COBOL applications is significantly higher than in other languages ($p < .01$). The median percent of highly complex objects for COBOL applications was just above 80%, while for all other languages it was below 20%. In particular, the median percent of highly complex components for the newer Object Oriented Technologies such as Java EE and .NET was below 5%, consistent with the objectives of object-oriented design.

**VII. SUMMARY AND NEXT STEPS**

Automated structural quality detection techniques are important to the Agile Methods community in several ways. First, automated analysis and measurement of structural quality should be incorporated into an automated continuous build environment as a component of an automated test suite. Second, the severity of this structural problems emerging from this analysis should be prioritized for immediate remediation or for inclusion as a structural story for a future sprint. Finally, the results of structural analysis provide a strong measure of technical debt, a critical management parameter for evaluating cost of application ownership and risk to the business.

A more complete report of these results appears in [4] and will be submitted for journal publication. An obvious follow-on research topic is to relate the internal quality results of applications to their operational performance and business impacts. Unfortunately none of the companies that provided these applications had collected the type of data that would make this outcome research possible. We are working with several companies to collect the outcome data needed to validate the expected relations between internal quality characteristics and application risks and costs.

**VIII. REFERENCES**


